Pedro Pinto, Tiago Carvalho, João Bispo, Miguel António Ramalho, João M. P. Cardoso. **"Aspect Composition for Multiple Target Languages using LARA,"** in *Computer Languages, Systems and Structures*, Elsevier, Vol. 53, Sept. 2018, Pages 1-26.

The published version is available online at: https://doi.org/10.1016/j.cl.2017.12.003

Aspect Composition for Multiple Target Languages using LARA

Pedro Pinto*, Tiago Carvalho, João Bispo, Miguel António Ramalho, João M. P. Cardoso

Department of Informatics Engineering
Faculty of Engineering, University of Porto

Abstract

Usually, Aspect-Oriented Programming (AOP) languages are an extension of a specific target programming language (e.g., AspectJ for JAVA and AspectC++ for C++). Although providing AOP support with target language extensions may ease the adoption of an approach, it may impose constraints related with constructs and semantics. Furthermore, by tightly coupling the AOP language to the target language the reuse potential of many aspects, especially the ones regarding non-functional requirements, is lost. LARA is a domain-specific language inspired by AOP concepts, having the specification of source-to-source transformations as one of its main goals. LARA has been designed to be, as much as possible, independent of the target language and to provide constructs and semantics that ease the definition of concerns, especially related to non-functional requirements. In this paper we propose techniques to overcome some of the challenges presented by a multilanguage approach to AOP of cross-cutting concerns focused on non-functional requirements and applied through the use of a weaving process. The techniques mainly focus on providing well-defined library interfaces that can have concrete implementations for each supported target language. The developer uses an agnostic interface and the weaver provides a specific implementation for the target language. We evaluate our approach using 8 concerns with varying levels of language agnosticism that support 4 target languages (C, C++, JAVA and MATLAB) and show that the proposed techniques contribute to more concise LARA aspects, high reuse of aspects, and to significant effort reductions when developing weavers for new imperative, object-oriented programming languages.

Keywords: LARA, AOP, Aspect Composition, Aspect Modularity, Aspect Reuse, Language Agnostic

1. Introduction

Aspect-Oriented Programming (AOP) [1] is a paradigm that aims at increasing program modularity by specifying code related with crosscutting concerns (e.g., logging, profiling, autotuning) into separate entities called aspects. It is common for an AOP approach to be focused on a specific target language (e.g., AspectJ [2] for JAVA and AspectC++ [3] for C++). When moving crosscutting concerns to an aspect, we are effectively moving code from a source file to an aspect file. As such, for seamless integration between

*Corresponding author
Email address: p.pinto@fe.up.pt (Pedro Pinto)

aspect and application code, we need access and support to language features. In fact, most well-known AOP languages are extensions of their target language [4]. The most relevant benefits from this approach are being able to specify additional behavior transparently in the aspect language, and a possibly lower learning curve for aspect developers, since they should be familiar with the target language.

Despite the benefits provided by such approaches, coupling an AOP language to its target language can present some drawbacks. For instance, any restrictions and limitations of the target language may be propagated to the AOP approach (e.g., AspectC++ [3], which extends C++, inherits its limited reflection capabilities). Moreover, tying such an approach to its target language may prevent opportunities for tooling reuse between AOP approaches of different target languages. Therefore, developing an aspect-oriented approach for a new language from the ground up is non-trivial and a significant undertaking, since common parts between language-specific AOP approaches that can be reused are few or non-existent.

We propose these two problems can be solved, or significantly reduced, if we adopt an approach that is, as much as possible, agnostic to the target language. On one hand, it allows the development of an AOP language with features that are independent of the target language; on the other hand, it enables tooling reuse, which can significantly reduce the effort needed to support new target languages. This can include compilers and/or interpreters for the AOP approach in addition to a well-defined API. Since aspects may be defined independently of the target language and the underlying framework may be the same, there are also new opportunities to explore, such as the possibility to reuse aspect code between different target languages, and the development of aspect libraries that support more than one target language.

LARA [5] is a Domain-Specific Language (DSL) for source-to-source transformations and analysis, inspired by AOP concepts [1]. LARA explores the idea that it is possible to have a single programming language, agnostic to the target language, capable of selecting points of interest and expressing source code transformations. To apply LARA aspects we use *weavers*, tools that translate the abstract concerns described in LARA to a concrete language.

However, a multilanguage aspect-oriented approach presents new challenges. The first challenge is how to specify, in a target-independent way, queries of specific points in the code. Previous work on the LARA language [5, 6] provides a solution for this first challenge, which is presented in Section 2.1. The second challenge is defining additional behavior for the application in a language-independent way. One of the ways LARA currently adds additional behavior is to allow insertions of arbitrary strings around the points of interest it captures. Figure 1 shows a LARA aspect that modifies source code to log function calls. Line 3 queries the code and selects all calls inside functions (i.e., `select function.call end`). The join point `function` represents the code for a function while the join point `call` represents the code for a function call. To this selection, we apply the rule inside the `apply` block (lines 4-8), which inserts the code inside the brackets (i.e., `%{}%`) before all captured calls. The inserted code prints the name of function where the call happened (i.e., `$function.name`), and the name of the called function (i.e., `$call.name`).

Although a powerful mechanism, raw code insertion can also become complex and error prone. For

```
aspectdef LogCall

 select function.call end
 apply
  $call.insert before %{
   printf("[[$function.name]]->[[$call.name]]\n");
  }%;
 end
end
```

Figure 1: An example of a LARA aspect that inserts a *print* instruction before function calls.

instance, syntax verification of the inserted code during aspect compilation is not guaranteed and is dependent on the implementation of the tool. Also, this example is not inserting the necessary includes for the function `printf` (i.e., `<stdio.h>`). In this paper, we build upon our current solution and present techniques to better overcome the challenge of specifying additional behavior in a language-independent way.

The main contribution of this work is a multilanguage[1] AOP approach that enables weaving of reusable advices, at the aspect and language level. More specifically, this paper addresses the following points:

- A comprehensive and updated description of the LARA framework and its components;
- The Weaver Generator, a tool that provides a base implementation of a weaver based on an initial Language Specification;
- A systematic classification of pointcuts and advices regarding reuse and composition;
- A set of techniques using the LARA framework for reuse and composition: generic weaver libraries, user library bundles and overlapping Language Specifications;
- A set of language-independent aspect libraries and examples for several crosscutting concerns that apply the presented techniques;
- Evaluation of our approach regarding the impact of the techniques in aspect code and the effort of developing a new weaver using the LARA framework.

Furthermore, the techniques that lead to our multilanguage support also result in improved code reuse, both at the weaver level and at the aspect level. We consider the developed aspects become more concise and are less likely to produce incorrect code, since in most cases the user can abstract from writing code in the target language.

This work is an extension of the paper *LARA as a Language-Independent Aspect-Oriented Programming Approach* [7], presented in SAC'17. This paper was extended and improved in multiple ways. There is a new

[1]Please note that the target languages of our weavers are imperative, possibly Object-Oriented, programming languages.

section describing the LARA framework and how to use it to develop a new weaver. Another new section was added where we present several possibilities for aspect reuse and composition. Since the submission of the previous work, we developed a new type of interface, *bundles*, which provide the users with library building features. This was integrated in the description of our approach and in the evaluation. One of the weavers presented in the previous paper, which targets C, was replaced by a newer weaver, which targets both C and C++. The examples in the previous paper were replaced with 8 new examples targeting 4 languages, and their evaluation section was augmented and updated. The related work section was heavily restructured and augmented in order to provide better comparisons across several dimensions.

The remainder of this paper is organized as follows. In Section 2, we present the LARA framework, its components and how it can be used to develop new weaving environments. Then, we enumerate possible options for aspect reuse and composition in Section 3, and, in Section 4, we describe how our approach implements these concepts, based on features of the framework and further techniques to improve aspect modularity and multilanguage support. Next, we discuss in Section 5, the results obtained when evaluating the developed techniques and the estimated effort of developing a new weaver. Finally, in Section 6 we present and review related work, and we conclude the paper, drawing our main conclusions in Section 7.

2. The LARA Language and Framework

Unlike many AOP approaches, the LARA-based approach was designed to be, as much as possible, agnostic to the target language and to specify insertions and code transformations for any supported target code. This was achieved by decoupling LARA from the target language model, which is a specification of the points of interest of the target language, as well as their attributes. When using LARA code to transform the input source code in a specific target language we need a *weaver*, which connects the language model and the target code representation, e.g., an Abstract Syntax Tree (AST).

The language model, named *Language Specification* in our approach, has three main components, which are currently specified in XML files. First, the *join point model* specifies which code structures a weaver can capture. Furthermore, in LARA, *join points* have relations between them, based on which points can be selected from others. For instance, we can select functions from within files and this is defined in the join point model. Then, the *attribute model* specifies which attributes are available in each join point, which can be as simple as the name of a function, or include semantic information, such as the type of a variable or number of iterations of a loop. Finally, the *action model* defines weaver-specific actions, which are used to advise the code (e.g., code insertions or loop transformations).

Figure 2 shows the general structure of a weaver based on the LARA framework. The *LARA Engine*, as all the components in the framework, is developed in JAVA. It is a generic component, which can be used by all LARA weavers and contains a compiler and interpreter for the LARA language. It needs a Language Specification to know which points in the code are available (*join points*). When instructed by the LARA Engine about weaving-related operations, the *Weaving Engine* queries and transforms the internal

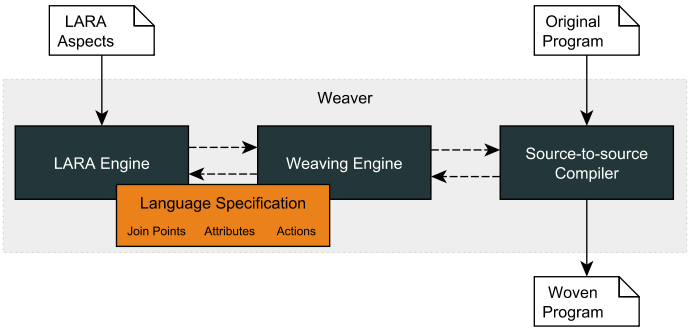

Figure 2: The general structure of a weaver based on the LARA framework.

representation (IR) of the *Source-to-Source Compiler*. The compiler parses the input code and builds the IR, which is changed according to LARA aspects. At the end of execution this component also generates back source code representing the altered program. We used source-to-source compilers as an example since most of our tools (and all presented here) follow a source-to-source compilation flow. However, this is not mandatory and a new weaver can have back-ends targeting multiple platforms, in source, binary or bytecode form.

LARA considers two separate groups of developers: weaver developers and aspect developers. This separation is similar to *Stephanies* and *Joes* introduced by August et al. [8]. The weaver developers (Stephanies) are few, but their work enables AOP approaches for new languages that can be used by many aspect developers (Joes). Additionally, the effort spent on the side of the weaver developer has the potential to multiply its payoff, in proportion to the number of users of the weaver.

2.1. The LARA Language

LARA [5] is a domain-specific language, inspired by AOP concepts [1], for programming strategies mostly regarding non-functional concerns addressed by code transformations, monitoring and instrumentation. Strategies are sets of related aspects that provide support for weaving a number of related concerns. LARA provides semantics that allow to query and modify points of interest in the target source code, and supports arbitrary JavaScript code to provide general-purpose computation (very useful for programming advices based on application properties).

This section presents a brief overview of the LARA language. For more detailed information please refer to previous work [5, 6]. Syntactically, LARA aspects have six main keywords, `select`, `apply`, `condition`, `exec`, `insert` and `def`. The first two can be seen in the aspect presented in Figure 1. These two keywords are used to *select* points of interest in the code and *apply* actions over them, i.e., they are used to specify pointcut expressions and advices, respectively. A `select` statement in LARA (line 3) is commonly known as a *pointcut* definition in AOP literature [1]. In a select statement one can define *pointcut expressions* by means of *join points* and optional filters based on the attributes of each join point. The pointcut expression is a chain of join points that follows the hierarchy defined in the join point model. This means that a join

point in a chain can only be *selected* if the previous join point can *select* it. The *apply* block (lines 4-8) is used to *advice* a join point [1] (we note here that the join points in the programming code are known as *join point shadows* [9], but for simplicity we refer here to them as join points). Inside this block, users act over the selected join points using weaver-defined actions.

The `condition` keyword is used to declare a block of conditions that must hold true for the advice to be applied, i.e., to further filter the set of join points on which an advice will act. It is common to use join point attributes to perform this filtering operation. The `exec` keyword is used inside `apply` blocks to execute weaver-specific actions over a join point. LARA has two default actions, `def` and `insert`, which have their own specific syntax and are used to define the value of a join point attribute and to inject code in the program, respectively. All other actions are added by the weaver developer and are executed with the `exec` keyword. An example can be seen in Figure 13, where the `DeclareVariable` action is used to declare a variable of a provided type in the selected scope.

The LARA language follows a simple declarative model using the presented keywords to specify pointcut expressions and advices. However, some strategies require more complex computations and an imperative programming model has been selected. For those cases, LARA adopted the syntax and semantics of the JavaScript programming language.

LARA strategies are composed of modular units, known as aspect definitions or `aspectdef`s. They are independent modules that can be called from within other `aspectdef`s, receive parameters and return outputs. They can be defined in separate files and can be imported as libraries, at the file granularity. This is performed with the `import` keyword at the top of an aspect file, and makes all `aspectdef`s inside the imported file available in the current aspect file.

LARA has some different concepts compared to common AOP approaches in terms of join point types, pointcut definition and advising format [10]. LARA join points are usually structural, due to the source-to-source weaving nature in which LARA was designed. This means that the join points that can be selected in a pointcut expression are generally syntactic, part of the program structure. For instance, existing LARA weavers do not have a function *execution* join point (as in other common approaches [2, 3]). However, they allow the selection of calls to a function or entry and exit points of that function, which can be used to provide these missing behavioral join points. In the context of this work, LARA only supports statically known join point information (called *attributes*, in LARA). For instance, it is possible to know the control variable of a loop but it is not always possible to statically know the number of iterations of that loop.

LARA uses scoping pointcuts [10] defined as join point chains that follow a hierarchical join point selection. It respects the join point hierarchy model (code structure) and can access static information (e.g. `function.body.call{name=="getX"}`), hence no dynamic conditions can be achieved with a LARA pointcut, as presented in this paper. Type patterns are available in LARA by means of select filters, boolean expressions that compare attributes of a join point to the intended values. It is possible to filter a join point based on specific conditional matching of attributes as well (e.g., regular expressions matching). Further-

more, one can also use composition of pointcut expressions by using `select` operators, e.g., the join operator (`::`) performs a *natural join*.

In the context of the work described in this paper, advices in LARA are executed at weaving time, advising the selected join points (more precisely, join point shadows) iteratively, as specified by the execution of actions. The actions that each weaver provides are the real pointcut actuators, and can be divided into structural and behavioral actions [10]. Structural actions are usually known as introductions [10] and intend to add new code members, such as variables or functions, or to extend class functionality (e.g. add new interfaces). On the other hand, behavioral actions are actuators that will add behavior to the program, whether by native code injection or by code transformations.

2.2. Supported Types of Pointcut Definition

LARA supports several types of join points (e.g., syntactic, semantic, execution). Most of the join points implemented in our weavers are of the syntactic and semantic kind, since they better fit our approach (i.e., source-to-source transformations), however LARA does not force a specific model, and it is entirely possible to develop a LARA weaver mostly based on execution events, e.g., as in AspectJ. Below, we present a list of common types of pointcuts, their description and if (and how) they are supported in LARA. We also present some comparisons to AspectJ since it is the most well-known AOP approach and its semantics have been used by many other approaches.

2.2.1. Syntactic Pointcuts

Syntactic pointcuts refer to static elements in the source code that use join points such as classes, functions or loops. Approaches such as AspectJ can interact with some syntactic elements, in what is called *Introductions*, to change the program's hierarchical structure.

As mentioned before, most of our aspects written in LARA use syntactic join points. Since most aspects for our use cases are concerned with elements of the source code, the join point models of the weavers we developed usually implement syntactic join points.

For instance, Figure 1 captures and changes any code location matching a function call, while also capturing the encapsulating function. Then, the name of both the function being called and the surrounding function (attributes of those join points) are used in the code that is inserted before the call. This simple example uses only static information that can be collected from source code analysis.

2.2.2. Execution Pointcuts

These pointcuts refer to events that occur during program execution, such as calling a function or setting the value of a field. These pointcuts are common in AspectJ and in other approaches that follow a similar model. Below is an example of an AspectJ pointcut expression, which captures assignments to a field called `age` from class `Person`:

```
pointcut setAgePointcut(Integer newValue) : set(* Person.age) && args(newValue);
```

In our weavers, such pointcut expressions are usually built using syntactic join points. For instance, the following LARA code, which uses only syntactic elements, can be used to emulate the AspectJ code above:

```
setAgePointcut : select var{name=='age', isField==true, reference=='write', declarator=='Person'} end
```

In this query, we select variable references and filter those by their name, whether they are a field, whether the reference is a write to that variable and by their declarator. In the case of a field, the declarator is the name of the class the field belongs to. This is an example on how syntactic constructs can be used to capture runtime events.

Alternatively, we could add the `set` join point, with the same name and semantics as in AspectJ, to the join point model exposed by the weaver. Thus, we would have a mixed join point model with both syntactic and execution join points. The resulting pointcut expression would be:

```
setAgePointcut : select set{name=='age', class=='Person'} end
```

This query looks for all field sets and filters them by field name, and parent class, using the attributes of the `set` join point. This results in shorter and more readable code, since most of the work is performed inside the weaver, looking for assignments to fields.

2.2.3. Semantic Pointcuts

Semantic pointcuts are those that are based on the meaning of the source code, and require a more refined analysis of the underlying AST. These are commonly present in LARA aspects. For instance, information such as the type of variables or loop iteration counts is often used to filter selected join points.

The following example selects loops with a small number of iterations as candidates for loop transformations, such as a full loop unrolling:

```
select loop{isInnermost==true, iterationCount<=32} end
```

This LARA code will select all loops and then filter out non-innermost ones as well as loops with a number of iterations greater than 32. This information needs to be calculated from the AST using the initial value, stopping condition and step of the loop. Keep in mind this operation will not always be available, based on static information alone. In such cases, the attribute will return the value `undefined`. On the other hand, if this information is available, it can be used for instance to find good candidates for parallelization, where we look for outermost loops with a large number of iterations. Loop transformations and general program optimizations are one of the areas where LARA has been used frequently, making these pointcuts necessary.

A more complex example, from ANTAREX project[2], uses LARA and semantic pointcuts to find implicit casts inside kernels of HPC applications. With this information, and the ability to change the types of variable declarations, it is possible to write strategies to avoid some of these casts and improve the execution time of the application kernels.

2.2.4. Control Flow Pointcuts

Control flow pointcuts match control flow events, for instance, if a certain function is currently in the call stack. While this type of pointcut is commonly used in AspectJ, LARA does not directly support it, nor is it part of any join point model presented in this paper (mostly due to our static source-to-source approach).

However, it is possible to implement flow-based pointcuts by using low-level code insertions that can be encapsulated in LARA libraries and used by other LARA aspects. The join point models used in LARA are rich enough to provide mechanisms to specify control flow pointcuts similar to AspectJ and/or other custom or more generic possibilities (e.g., look to a sequence of code blocks considering basic blocks).

The example below shows a possible LARA library (`ControlFlow`) that provides similar functionality to that provided by AspectJ's `cflow`. It is based on the example in Figure 1, and in this case uses the control flow library as a way to restrict the application of the advice code:

```
var cflow = new ControlFlow($program);
// ...
$call.insert before clow.cflowbelow($targetFunction,
 %{
  printf("[[$function.name]]->[[$call.name]]\n");
 }%
);
```

The code inserted by the user remains the same, and the changes consist in the use of the control flow library and its `cflowbelow` method. A possible implementation is for the library to automatically add the code needed to maintain a call stack, when the `cflow` variable is instantiated (the variable `$program` is a previously selected join point that represents the current program). The function `cflowbelow` could wrap the code provided by the user with code that checks the call stack and tests whether the target function is on the current frame but not the one being currently executed. In this example, this target function is represented by `$targetFunction`, a function join point that was selected beforehand by the user. A difference to AspectJ's equivalent pointcut is that in this example we are clearly selecting calls that appear bellow the target function and this is all we interact with. With AspectJ's `cflowbelow`, all join points below the provided target join point are automatically matched.

Another example is the `within` pointcut used in AspectJ, which matches when the executing code belongs to a certain class. Note that the semantics of the pointcut `within` will differ according to the target language,

[2]http://www.antarex-project.eu/

the main modular unit in JAVA is the class, but in C or MATLAB is the file. Consider the code below, where the code in Figure 1 is modified to include the equivalent of a `within` pointcut.

```
// ...
  select file{name == targetFile}.function.call end
  apply
    $call.insert before %{
     printf("[[$function.name]]->[[$call.name]]\n");
    }%;
  end
// ...
```

AspectJ's concept of `within` is already contained in the join point chain that is used for LARA pointcut expressions. In this example we can filter the matched function calls by explicitly defining in which file the calls should be. Since this is a C example, we are using the file as the modular unit, as opposed to JAVA, where we would likely use classes. AspectJ's concept of `withincode` is also covered if we use the attributes of the `function` join point to filter the pointcut.

As a final note, since the concepts of a control flow library are generic across multiple target languages, the library can be designed with a language-independent interface, allowing each weaver to provide its own implementation, bound to its target language. More information on this kind of libraries is presented in Section 3.2.1 and in Section 4.2.

2.3. Compiler and Interpreter

The LARA compiler (LARAC) converts LARA aspects into an intermediate representation (Aspect-IR) based on the provided target language model [5]. LARAC is responsible for handling usual compile-time verifications, such as existence of called aspects and the existence of a `select` for an `apply`, as well as validating the aspects according to the Language Specification. The latter task includes pointcut validation (e.g., checking if the join point exists in the Language Specification) and action validation (e.g., checking whether action arguments are valid). After compilation, a weaving environment can take the generated Aspect-IR and interpret the aspects and apply actions to the target code.

The LARA interpreter (LARAI) has been developed to simplify integration of the LARA language on different weaving environments. LARAI executes most of the code defined in a LARA file (e.g., loop and conditional statements), the weaver being responsible for performing weaving-related tasks. A standalone version of the interpreter has been used for different purposes [11]: as a scripting language based on the JAVASCRIPT syntax, to execute external tools, to coordinate compilation flows (e.g., instrumentation → optimization → compilation → execution → profiling) or to define design-space exploration schemes [12]. However, as a standalone tool, the interpreter cannot carry out weaving-related tasks, such as selecting points in the code and applying actions. In order to do so, LARAI must be connected to a Weaving Engine that is responsible for building an IR (e.g., AST) for the target application, select join points, retrieve attribute information, apply actions, and generate the modified code.

To ease development of new weaving environments, LARAI provides a JAVA API for Weaving Engines, which works as a bridge between the LARA interpreter and the IR of the target application. This interface reduces the effort needed to develop a new weaver and requires the implementation of abstract classes by the weaver developer:

- WeaverEngine: abstract class representing the Weaving Engine, which includes, among other facilities, `begin` and `close` methods which the interpreter uses to start and end the weaving process.
- JoinPoint classes: the join point abstraction classes represent join point instances, and there is one for each type of join point (e.g., `file` or `function`). They provide methods to select other join points, to retrieve attributes and to apply actions.

As an example of the level of framework reuse it is possible to achieve with LARAI, consider the attribute-based filters that can be defined in a `select` statement for each join point (e.g., see Figure 6). The weaver developer only needs to implement the methods that return each join point type and each join point attribute, being the responsibility of LARAI to request the required join points, attributes and perform the filtering. Additionally, this API allows LARAI to provide out-of-the-box, to all weavers, a customized integrated development environment that contains a LARA aspect editor with syntax checker and integrated controls for weaving actions, as well as a language specification guide.

2.4. The Weaver Generator

Even with the LARAI API, developing a Weaving Engine from the ground up can still require a considerable amount of effort, mostly due to the fact that each join point in the Language Specification requires a class, each with code that connects the Weaver Engine to the interpreter. Additionally, manually maintaining the Weaver Engine in sync with the Language Specification join points, select relations, attributes and actions is too costly and error prone.

Fortunately, a significant part of this effort can be automated. The Language Specification and LARAI weaver-related interactions have a close association, as all the possible weaving requests are described in the specification. To complement the weaving API provided by LARAI, we have developed the Weaver Generator which generates the skeleton of a new Weaving Engine from a Language Specification. This generator allows quicker development of new weaving environments, as the developer just needs to implement the abstractions provided by the skeleton. Although the number of generated classes that have to be extended is similar to the number of classes that had to be manually created, the generator implements all required infrastructure and interfacing code. All join points are generated as abstract classes, and their selects, attributes and actions are added to the class as abstract methods. The abstract class guarantees that the join point implementation conforms to the description in the Language Specification, as it forces the weaver developer to implement the attributes and actions specified. The work left for the developer is essentially focused on writing code

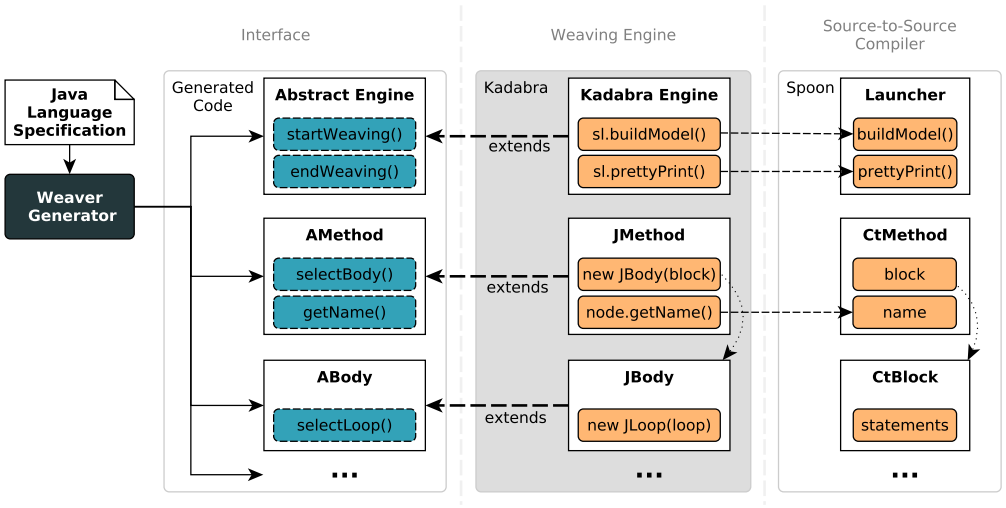

Figure 3: Representations of the code generated by the Weaver Generator (Interface), the code developed by the weaver developer (Weaving Engine), and the code of the source-to-source compiler .

for the abstract methods, which would consist in accessing the IR for selection, attribute queries and action purposes, working directly on the chosen IR.

As an example, Figure 3 shows a small part of the Kadabra weaver, which targets the JAVA programming language. To develop the weaver, the Language Specification is given to the Weaver Generator, which generates the abstract classes (see the *Interface* area). Kadabra uses an existing JAVA-to-JAVA compiler, Spoon [13], which builds the IR for JAVA code (see the *Source-to-Source compiler* area). The work of the weaver developer is to implement the Weaving Engine component and the concrete join point classes (see the *Weaving Engine* area), which bridges the generated interface and Spoon. In the example, this effort corresponds to the development of the concrete classes `KadabraEngine`, `JMethod` and `JBody`, which extend the automatically generated abstract classes.

The Weaver Generator allows weavers to be developed incrementally. The Language Specification does not have to be completely defined when the weaver is initially developed. Whenever required, the Language Specification can be upgraded with new join points, attributes and actions and given again to the generator. New classes are added to the Weaving Engine and existing classes are updated with new selections, attributes and actions. Since the Weaver Generator works at the abstract classes level, it does not conflict with the code implemented by the weaver developer.

Figure 4, on the left side, shows part of the Language Specification for a weaver. The specification is divided into three models, specified as XML files: join point hierarchy, artifacts and actions. The join point hierarchy defines available join points and how to select them. The example declares a join point named `function`, which is able to select its parameters (which are `var` join points) and body (the declaration of the `var` and `body` join points is omitted in the figure). We can use aliases when we intend to give a specific meaning to the type of join point we want to select. For instance, in this example we use the alias `param`

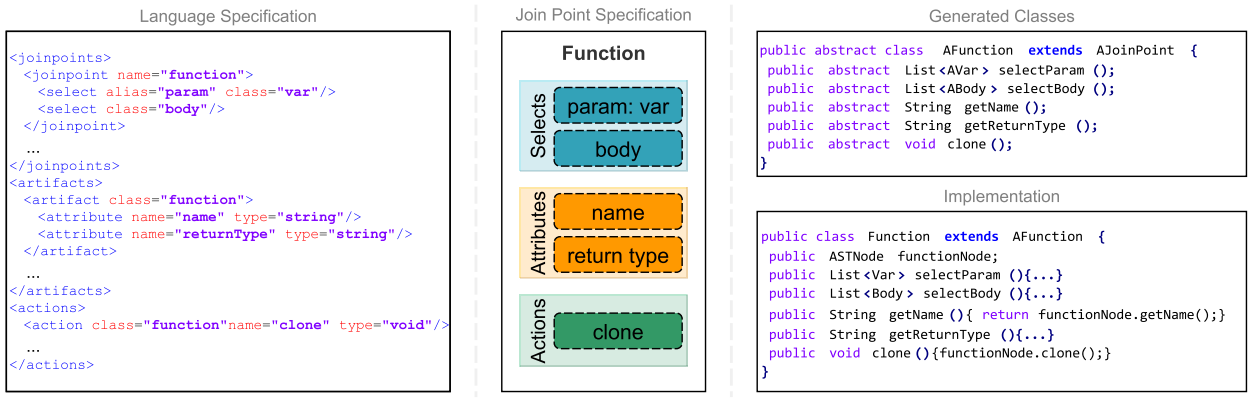

Figure 4: Example of a join point definition in a Language Specification: the join point function and its selectable join points, attributes and actions.

to specify that we want the parameters of the code, but they are internally join points of type `var`. The artifacts model defines the attributes of a join point, such as function name and return type. The actions model defines the actions that can be applied to the join points.

Based on the Language Specification, the generator creates the abstract classes, and the weaver developer implements the concrete classes. The class `Function` shows the bridge between the LARA interpreter and the source-to-source compiler as it uses a node from the AST of the compiler to get attributes or to apply the `clone` action.

2.5. Mapping Between Join Points and Input Applications

Based on the general structure of a weaver presented in Figure 2, consider an example weaver for `C`, such as the one presented in Figure 5. The figure shows the weaving process of LARA and how the aspect is mapped to the input C application. The join points defined in the Language Specification are understood by the LARA Engine and the Weaving Engine and they communicate with an interface based on this specification.

In a LARA aspect, `select` and `apply` blocks use join point information (structure, attributes and actions) in order to define pointcuts and advice, mapping user strategies to the join points of the weaver and operations on those points. On the other side, a compiler parses the source code of a C application and builds its Intermediate Representation (IR), in this case an Abstract Syntax Tree (AST), mapping the input application into a structure that can be changed during weaving. Finally, in the core of the weaver, the Weaving Engine translates join points into actual IR elements, as well as user intentions into concrete changes in the AST. In this example, the join points representing programs, files, functions and loops are translated to concrete AST nodes, *Program*, *Translation Unit*, *Procedure* and *ForLoop*, respectively.

We note that the Language Specification does not need to provide a one-to-one mapping to the target language or selected IR. Instead, it needs only to specify which parts are considered points of interest and

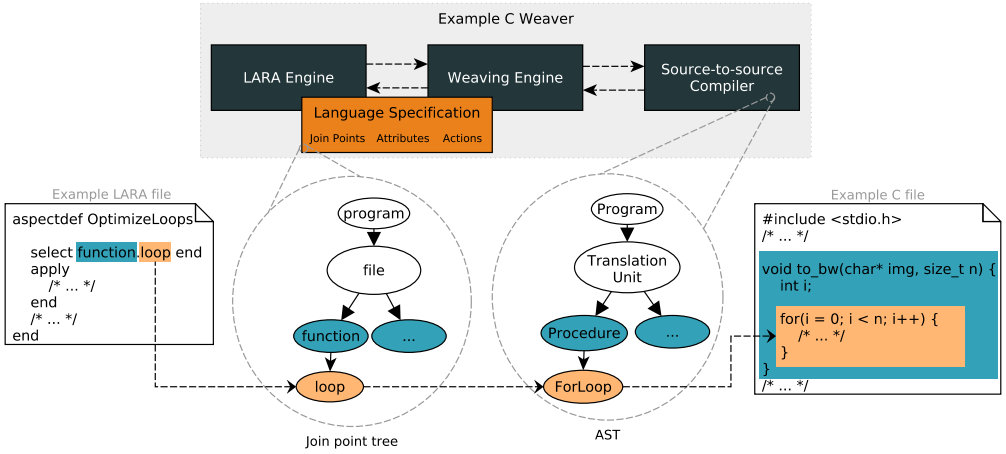

Figure 5: An example C weaver and the mappings between aspects, join points, the internal compiler representation and the source code.

which attributes they have.

3. Composable and Reusable AOP Approaches

This section presents the main features we consider are needed to support aspect composition and reuse, as well as multilanguage capabilities. We describe possible implementations for each feature, dividing the features into two groups: pointcut features and advice features.

3.1. Pointcut

In this section we present mechanisms to improve composition, reuse and multilanguage support at the pointcut level.

3.1.1. Generic Syntax for Pointcut Expressions

Pointcuts can benefit from a mechanism that allows defining points in the code in a way that is agnostic to the target language. For instance, SourceWeave.NET [14] uses XML to specify pointcut expressions for .NET languages. In our approach, the structure of the LARA language remains the same, regardless of the target language. Figure 6 shows three examples of *select* statements, which define pointcuts in LARA (see Section 2.1 for more details about the syntax of the language). When writing an aspect for another language, the names of the join points and the names of their attributes may change, but the syntax of the select mechanism remains the same, agnostic to the underlying target language. For LARA, this mechanism can help users write aspects for different target languages since the syntax is the same and, when languages share a common join point model, even use the same select expressions.

```
// selecting all declarations of variables
select vardecl end
// ...

// selecting a specific function (based on the name)
select function{name == 'kernel'} end
// ...

// selecting an innermost loop inside a specific function (based on the name)
select function{'kernel'}.loop{isInnermost == true} end
// ...
```

Figure 6: An example of the mechanism used by LARA to capture join points. The syntax remains the same regardless of the join points being selected.

3.1.2. Common Join Point Model

Several languages share common structural and behavioral concepts. For instance, there are common syntactic elements across Java, Matlab and C++ (e.g., function, loop and statement are shared syntactic constructs). Similarly, one can also identify common behavioral concepts, such as function calls, or getting and setting values of variables or fields. It is possible to identify and capture such elements with pointcut expressions which can target multiple languages, assuming they all share these same elements.

The approach presented in this paper exploits the fact that our target languages share syntactic constructs, since they are all imperative languages. Each weaver has its own join point model but there is a common subset shared between all supported languages. However, this is not enforced by the LARA framework, but rather a consequence of our design choices for each weaver and for the framework. Because of this characteristic, it is possible to write pointcut expressions which can be used in languages such as Java and C++. In our current approach, a pointcut expression that uses a join point that is not available for the current language (i.e., outside of the shared subset) will result in an error saying the desired join point is not part of the model for that language. Compose* [15], a different approach with a focus on message passing, uses behavioral join points that are shared across multiple languages.

3.1.3. Advice Inheritance

This idea applies Object-Oriented Programming (OOP) concepts to AOP, namely the possibility of extending and overriding advices. With this idea we can reuse the same pointcut and apply different advices. For instance, consider a case where you intend to log assignments to fields of a specific class, but you don't want to specify how to perform the logging operations. You could define your point and mark the advice as abstract. Whoever intends to use that aspect needs to extend it and implement the advice, e.g., providing implementation to log to the console or to a log file. While this can be used as a way to enhance general reuse, it can also be a platform to support multiple target languages. If all our target languages share a join point model (e.g., AspectJ's execution model), we can extend the abstract aspect, reuse the pointcut

```
1  // main aspect definition which logs calls to 'setters' to a file
2  aspectdef Main
3   var fileLogger = new LogToFile();
4   call SettersAdvisor(fileLogger);
5  end
6
7  // an 'advisor' with a well defined pointcut that delegates the advice to the input aspectdef
8  aspectdef SettersAdvisor
9   input Actuator end
10  select methodCall{name ~= 'set'} end // ~= is the regex match operator
11  apply
12   call Actuator($methodCall);
13  end
14 end
15
16 // 'actuator' that logs to a file
17 aspectdef LogToFile
18  input $methodCall end
19  // ... code that logs to a file information about the $methodCall join point
20 end
21
22 // 'actuator' that logs to the console
23 aspectdef LogToConsole
24  input $methodCall end
25  // ... code that logs to the console information about the $methodCall join point
26 end
```

Figure 7: An example of LARA code that illustrates how to have clearly defined pointcut code which can be reused for multiple advices.

definition and simply override the advice with concrete implementations for each of the languages we intend to target.

We can emulate this feature (at least its practical consequences) even though LARA has no direct support for OOP applied to aspect definitions. In LARA, aspect definitions are first-class objects which can be passed as arguments to other aspect definitions. In order to emulate the results of advice inheritance, we first define multiple *actuator* aspects with only advice code that works on join points taken as parameters. Then, we have a single aspect definition which provides a concrete pointcut, our *selector*. This aspect takes another aspect as an input and calls it using the selected join points. At this point, we can execute this selector aspect with different actuator aspects, which will result in applying different modifications to the same pointcut. Figure 7 shows an example based on logging concerns that uses this technique. In the `Main` aspect definition we instantiate the advice we want (line 3). It is the file logger in the example but it could also be the console logger. Then, this instance is passed to `SettersAdvisor`, which has a well defined pointcut and that calls the logger on the selected join points. In this way, we can have a single, well-defined pointcut which can be reused for multiple advice definitions.

3.1.4. Aspect Multiversioning

The main idea of a multiversioning approach is that there is a way to signal (implicitly or explicitly) what is the current target language. This information is used to switch between multiple user-provided versions of the pointcut, each targeting a different language. When the target language changes, the pointcut expression is automatically chosen based on whatever mechanism is available.

We can identify a few mechanisms which can be used to signal the current language. The weaver can set a flag or attribute which can be explicitly checked in the aspect code and used, e.g., with conditional statements, to choose the correct pointcut. An alternative is the definition of multiple pointcut expressions which are marked with the language they are targeting, using either naming schemes or annotations. Finally, it is also possible to write these pointcut definitions in multiple files and follow a naming scheme or use a configuration file so that the weaver can automatically choose the correct version. These two latter approaches are implicit and transparent to the users, as they may just provide alternative pointcut versions and let the weaver pick them according to the target language.

While this may lead to a more verbose user definition of the pointcut, it may also lead to a more direct and more refined set of selected points. This is because, for each version, the developer can use language-specific features that would otherwise not be possible (e.g., when using a common join point model). Another advantage of this approach is that it may lead to less initial effort when developing a tool or framework that supports multiple languages. Such a tool can simply consist of multiple weavers *glued* together and a system that 1) decides which weaver to call based on the input language, and 2) extracts the relevant aspects from the multiversioned code developed by the user.

3.1.5. Merged Join Point Model

This idea is the opposite of the *Common Join Point Model*. We have different join point models for different languages, but instead of accepting a shared common subset, the pointcut expressions accept structures or behaviors that are language specific. To be more precise, the weaver would take the union of the sets of join points for all supported languages, rather than their intersection, as would happen in *Common Join Point Model*. The tool is responsible for checking the correct model for the current target language and decide whether or not the pointcut is valid.

Following this implementation means not raising an error when the join point is invalid, but rather fail gracefully (for this pointcut) or try to resume execution with available information. If this was implemented in the LARA framework, one of two possibilities would happen when the pointcut chain refers to a join point unavailable in the current language. The selection of that specific join point returns nothing and, if there is a selection chain, the chain is interrupted and nothing is advised. The alternative is that the specific join point is dropped from the chain and the tool tries to continue the execution from the next point in the chain.

3.2. Advice

In this section we present possible implementations to improve composition, reuse and multilanguage support at the advice level.

3.2.1. Interface

The idea of using interfaces relies on three concepts. First, the definition of aspect interfaces (conceptually similar to OOP interfaces) which provide a contract that any implementing weaver has to follow. Second, the definition of such contracts so that they describe common concerns at a high abstraction level. Finally, the development of aspect libraries implementing these interfaces in ways that are specific to some purpose or target language.

The main idea is to have a standard library of aspects which every conforming approach implements. In this way, the original definition presents the interface, which is the same regardless of which weaver is being used or which language is being targeted. Based on this, each weaver is responsible for implementing this interface for its target language. As an example, consider the act of logging a function call with native code insertions. After the selection of the call, we can insert the logging code around it, describing exactly which code is inserted (e.g., a call to `printf` in C or to `System.out.println` in JAVA). With an interface, we could call a `Logger` library that provides the aspect `log`, which takes a join point and the text to print, and the library implementation for that language automatically deals with inserting the logging code.

If concerns are specified at this proposed high level, it becomes easier to generate abstractions and let weavers deal with the details concerning specific languages. An added benefit of this approach is that we end up specifying our intents in a more declarative programming style, rather then just specifying how to directly change the code by insertions or any other means. As a downside, each specific concern might need its own library which needs to be implemented by the weaver developer, although this can be ameliorated if interfaces can be built on top of other interfaces.

3.2.2. Pointcut Inheritance

Much like the idea that was presented for pointcuts, this applies OOP concepts to AOP approaches. In this case, it would result in the ability to declare abstract aspect definitions or pointcuts, which can be extended. This enables reuse of predefined advices, since they can be applied to many different pointcuts, depending on how these pointcuts are concretely implemented. For instance, one can completely re-implement a pointcut and change the set of join points affected by the advice, or can simply refine an existing pointcut with further filtering, without changing the actions to be performed. An example of inheritance can be seen in AspectJ [2], in which we can have abstract aspect with one or more abstract pointcuts and a specific advice. Then, it is possible to extend those aspects and define concrete implementations of the pointcuts.

From the general idea that inheritance can lead to aspect reuse, it is possible to see how one can extend this idea to a paradigm that targets multiple languages. It is possible to have an abstract pointcut and associated advice and then implement the concrete pointcut for each of the target languages, since one

```
1  // selects all method calls whose name contains 'get' and calls an aspect definition with the advice
2  aspectdef LogGetters
3   select methodCall{name ~= 'get'} end
4   apply
5    call LogMethodCall($methodCall);
6   end
7  end
8
9  // selects all method calls whose name contains 'set' and calls an aspect definition with the advice
10 aspectdef LogSetters
11  select methodCall{name ~= 'set'} end
12  apply
13   call LogMethodCall($methodCall);
14  end
15 end
16
17 // receives a mathod call join point and inserts code to be executed before
18 aspectdef LogMethodCall
19  input $methodCall end
20  $methodCall.insert before 'System.out.println("Calling [[$methodCall.name]]");';
21 end
```

Figure 8: An example of LARA code that illustrates how to have clearly defined advice code which can be reused for multiple pointcuts.

knows how to select the desired points for each of those languages. Compared to the other OOP idea presented before, *Advice Inheritance*, this seems harder to implement, since it relies on an advice capable of targeting multiple languages, while the former needs only a common model of join points and an agnostic way of expressing the pointcut.

As an aside, while we do not have OOP features applied to aspect definitions in LARA, we can somewhat emulate the practical results of this concept, by taking an advice and simply plugging different pointcuts. This way, we change the set of points that are targeted by redefining only the pointcuts. To this end, we can have an aspect definition which has only the advice part implemented and that takes, as input, a join point (since they are treated as first-class objects in LARA). This means that any other aspect definition can redefine the pointcut and pass the selected points to the aspect that will advise the application. An example of this mechanism is shown in Figure 8. In this simple example, aspect definition *LogMethodCall* has the advice code, which will print the name of the methods right before they are called (lines 14–17). The other two aspect definitions each define their own pointcut and then pass the captured join points to *LogMethodCall*. So weaving either *LogGetters* or *LogSetters* will log calls to getters or setters, respectively, but the advice code is reused.

3.2.3. Aspect Multiversioning

This idea is similar to multiversioning applied to pointcuts. There is a mechanism that indicates, implicitly or explicitly, which language is being currently targeted. The user develops multiple versions of the advice and switches to the correct one based on a explicitly flag or lets the weaver take all versions and pick the correct one based on its implicit knowledge of the current target language.

An example of the multiversioning idea applied to advices can be seen in UniAspect [16], where annotations are used to identify pieces of advice code that are meant for specific languages. For instance, code inside the block `@Java{...}` is meant to advise JAVA programs, whereas code inside a `@C{...}` block would be targeting C programs. In this example, the user provides different versions inside the same source file, but a general approach would allow using different files as mentioned in the pointcut section.

3.2.4. Common Language for Advices

A final possibility to improve multilanguage support at the advice level would be to have advices specified in a common language, which abstracts the behavior from the target language. We consider two possible approaches: translation, execution, and code generation for specific concerns.

We can have generic advices that use a high-level language (e.g., a custom DSL) to write the code that specifies the behavior to be executed at/added to the join point. The weaving framework can provide a parser and an internal representation for this language, which each implementation has then to translate to the target-language. This path allows to specify advices in a completely agnostic way and provides some reuse of the underlying framework. On the other hand, designing a common language that can be translated to many other languages can impose many constraints and limits the applicability and potential evolution of this solution.

A second approach is to directly execute the advice written in a common language, instead of translating it to the target language. Consider an advice written in a language such as JAVASCRIPT. A weaver can insert hooks to a JAVASCRIPT interpreter that executes the advice code and provides the additional behavior. It can be a feasible solution for target languages such as C++ and JAVA, which have robust interpreters (namely, V8 and Nashorn). The downsides of this approach include higher execution overhead, when compared with execution of native code, and providing an interface between the original application and the advice execution engine.

4. LARA Approach

In this section we describe features and techniques we use in order to achieve modular and composable aspects, which in turn lead to more reusable aspects that may even target different languages. The techniques include LARA libraries, bundles of user code, and defining the join point model in a way that increases compatibility between languages, while maintaining most language features intact. The first technique is our implementation of *Interface* (Section 3.2.1) for the weaver side. The second technique is our implementation

of *Aspect Multiversioning* (Section 3.2.3) for the developer side, which also uses some concepts defined in *Interface*. Finally, the third technique is our implementation of *Common Join Point Model* (Section 3.1.2). From the language description presented in Section 2.1 we can also say our *select* statement is an implementation of *Generic Syntax for Pointcut Expressions* (Section 3.1.1). The benefits of these techniques can be harnessed across several levels, depending on where they can be implemented: at the LARA framework level (becomes available to all weavers), at the weaver developer level (becomes available to all weaver users) or at the weaver user level.

4.1. General Composition and Modularity Techniques

As explained in Section 2.1, LARA aspect definitions (or `aspectdef`s) are treated as independent modular units which can be called from within other `aspectdef`s. They can be parameterized as well and are generally treated as functions (they are callable, execute a set of instructions, including advices, and return output values). This helps separating sub-concerns of larger strategies, leaving individual `aspectdef`s to perform well-defined tasks, improving the modularity of the entire program and maintainability of aspect code.

For instance, imagine using LARA aspects to instrument code and collect runtime data for either execution time, energy consumption or memory usage. Whatever the case may be, users will likely also want to save those results to a file. Therefore, in addition to the concern that instruments the code, one also needs to log the resulting data, a concern which can be reused. Taking this idea a step further, in LARA aspects we can also import other compilation units with one or more `aspectdef`s, meaning the reused aspect code does not need to be in the same aspect compilation unit. This also means that users may develop their own aspect libraries, reuse them and distribute them.

These features, alongside the fact LARA treats join points and aspect definitions as first-class objects, allows us to make use of the techniques presented in Section 3. For instance, with the technique presented in Figure 7, we can have clearly defined pointcuts and then use several aspect definitions to add different behavior to the application, effectively promoting reuse at the pointcut level. Conversely, the technique presented in Figure 8 allows us to define additional behavior for a common concern once, and then use it for multiple different pointcuts, promoting reuse at the advice level.

The features presented in this section were already part of the LARA language [5, 6] at the time of this work. They are the building blocks that allow us to develop our ideas regarding aspect reuse and multilanguage support.

4.2. Generic Aspect Libraries

As already explained, LARA was developed as a modular AOP approach and supports the import of libraries. We can have very simple JavaScript-based libraries, called from anywhere inside an `aspectdef`, that are useful for simple tasks, such as code generation. Please see the example in Figure 1 and consider we have a JavaScript library, `CodeGen`, that provides the function `println`, which can generate source code for printing information. Lines 5-7 of Figure 1 can be replaced with:

```
var code = CodeGen.println("[[$function.name]]->[[$call.name]]");

$call.insert before '[[code]]';
```

The benefits of using code generation libraries are threefold. First, there is the obvious reuse of code. A developer can write the code generation library, test it and use or provide it to other developers. Secondly, an important benefit is that aspect developers may need to write less native code, letting the library do this for them. In turn, this will likely speed up aspect writing and make it less error prone, as well as also improve intelligibility. Finally, we can also think of having weaver developers write their own versions of such a library, in a way that targets their supported languages. This would transform this into a multilanguage library.

Most uses of LARA require more complex operations than this simple code generation operation. There is added complexity when defining pointcuts and, generally, the advice code to be weaved is considerably more complex than the previous example. However, even at this level, we can still find aspect definitions repeating the same concerns, meaning we could likely develop an abstraction and include those in a library. Aspect definition libraries are used when there are common tasks that query and modify the target code. Examples include monitoring, such as timing parts of the program (e.g., function and loop execution), and logging activities, such as error conditions and function calls.

Consider the example presented in Figure 9, in which two weavers (Clava[3] for C/C++ and MATISSE[4] for MATLAB) are applying the same concern over an application. For the sake of space, the original application input is not presented. In this example, assume we intend to log function calls before they happen. As can be seen in Figure 9(a), the aspects definitions are different because we need to insert different native code for each target language. However, if we abstract from the code that needs to be inserted and consider the high-level operation of logging, we can extract this concern and implement it as a library with a well-defined interface. Furthermore, weaver developers can implement this library for their own target language and provide them alongside their weavers, which is represented in Figure 9(b). In the example, both Clava and MATISSE have their own implementation of the logging library, which is distributed with the weaver. At this point, we can actually use the same exact `aspectdef` for both weavers/languages. By doing this, we not only improve aspect reuse, but also add support for multilanguage aspect libraries. This is the concept we present as *Generic Aspect Libraries* and that is an implementation of the concept of *Interface* that was presented in Section 3.

Although the example presents a case where we can use the same aspect definition for multiple languages, this may not always be the case. Possible reasons include differing join point models or even situations where the aspect developer is required to provide language-specific code to be inserted. Nonetheless, by

[3]https://specs.fe.up.pt/tools/clava
[4]https://specs.fe.up.pt/tools/matisse

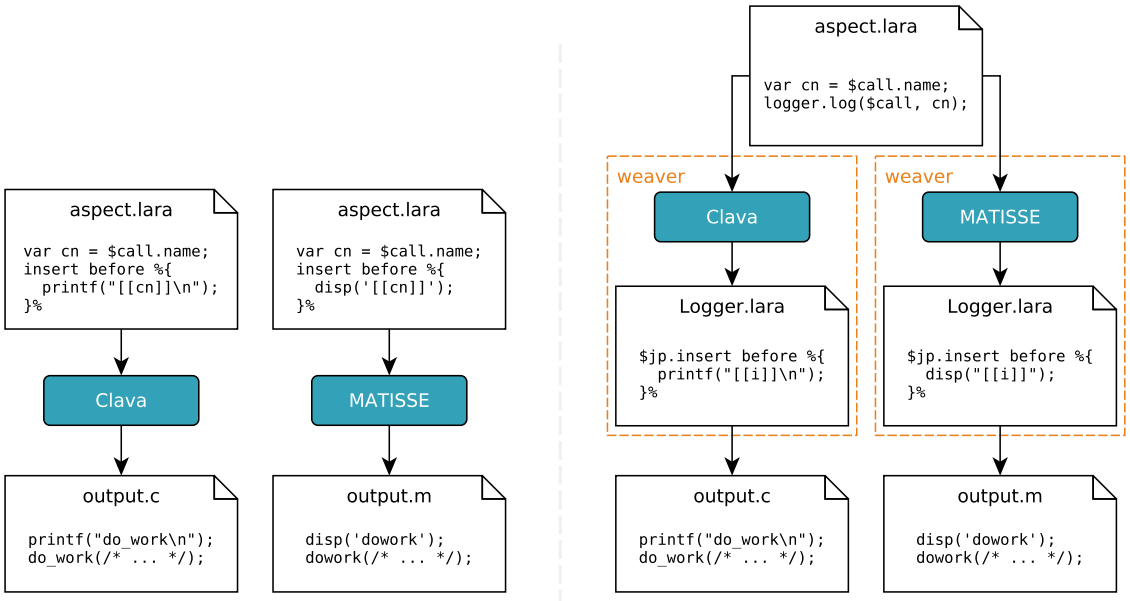

Figure 9: An example of a concern which can be abstracted to a high-level library. With this, it is possible to migrate from having an aspect for each target language (a) to having a single aspect that can be used for multiple languages (b).

using libraries, we still achieve the benefits presented earlier. More concretely, we improve reuse, readability and maintainability, and we reduce the need to write native code, making aspect coding less error-prone. Because these libraries have a very well-defined interface, we can move from one language to another and expect the same behavior.

For a more concrete example, see Figure 10, which shows a LARA aspect using the `Timer` library that can be used to perform simple timing measures of points in the code. Line 1 imports the library, and line 5 creates an instance of a timer library. Line 7 selects function calls and line 9 uses the timer instance to insert a timing measure around selected calls and a print of the result. When targeting languages like C or C++, this library automatically adds the needed include directives. Multiple weavers can provide their own implementation of this aspect library for their target language. This results in a generic library, which allows aspect developers to write strategies at a higher abstraction level. The LARA aspect presented in Figure 10 can be used to weave source code from multiple languages, as long as there are weavers targeting those languages that implement their version of the Timer library.

For an example input C program, a weaver like Clava would generate the code in Figure 11. Both setup code and timing/printing code are highlighted differently. The program now measures and prints the execution time of all function calls in the application, using native C code commonly applied for this task. As mentioned, the library automatically deals with inserting the needed directives so the aspect developer needs only to worry about describing the intents of the behavior to be added, rather than focusing on the

```
import Timer;

aspectdef TimeCalls
 // Create microseconds timer that prints information to the console
 var timer = new Timer("MICROSECONDS");
 // Time all calls
 select call end
 apply
  timer.time($call, "Time:");
 end
end
```

Figure 10: LARA aspect using a Timer library to measure execution times of all function calls.

details of how and where to insert code.

A further step is to standardize libraries between weavers. Since libraries can be developed at the weaver level, if two weavers share the same API (e.g., `Timer`) we can enable aspect code compatibility between weavers, at the library level. So far, we have started the implementation of three libraries that are part of our standard LARA library of aspects. For now, we have libraries to instrument an application and measure, around arbitrary code points inside a function, the time and energy consumption. Moreover, we also developed a library that is used to log information, either to the console or to a file. Any weaver developer wishing to conform to this standard needs to provide implementations of these libraries that follow the defined interface.

From the description of this technique, we can see these libraries are developed at the weaver side. That is, weaver developers provide their implementations of the library interfaces for their specific weavers/languages. *Bundles* are a technique that intends to provide the same benefits to our end users, i.e., the ones writing aspects and using weavers.

4.3. Bundles

Aspect library bundles exist so that end users can also develop their multilanguage aspect libraries and distribute them. This is a compromise between the concepts of *Interface* and *Aspect Multiversioning* from Section 3, since a user provides a library interface with corresponding implementations, but also marks each version as targeting a specific weaver. The weaver then reads all needed files and imports the correct ones. This process is transparent to the end user that just intends to use a library distributed as a bundle.

Figure 12 exemplifies how a bundle can be used. In this example, let us assume the user wants to write an aspect that instruments an application in order to generate a dynamic call graph of its execution. For the user writing an aspect, in (a), the only needed action is adding an `import` statement, as if any other library was being imported. When executing the weaver with the input application and the aspects to be weaved, the user also configures the weaver to include the folder containing the bundle. For a user wanting to distribute a bundle, in (b), we need a directory with subdirectories targeting all supported weavers (one per weaver),

```c
#define _POSIX_C_SOURCE 199309L
#include <time.h>
#include <stdio.h>

double bar() {
   // ...
}

double foo() {
 // ...
    struct timespec ct_start_0, ct_end_0;
    clock_gettime(CLOCK_MONOTONIC, &ct_start_0);
    a += bar();
    clock_gettime(CLOCK_MONOTONIC, &ct_end_0);
    double ct_duration_0 = ((ct_end_0.tv_sec + ((double) ct_end_0.tv_nsec / 1000000000)) -
    (ct_start_0.tv_sec + ((double) ct_start_0.tv_nsec / 1000000000))) * (1000000);
    printf("Time:%fus\n", ct_duration_0);
 // ...
}
```

Figure 11: The resulting C code when applying the aspect presented in Figure 10. We highlight the code inserted for setup and timing and printing calls.

as well as a configuration file (`lara.bundle`) which tells the system how to import the files. Additionally, a user developing a library bundle can also provide a `lara` subdirectory, which is always included regardless of which weaver is being used. This mechanism provides an efficient way for the user to develop and import aspect code shared between different implementations.

Because these libraries are developed and provided by users, the interface to the library can be different depending on the language. As opposed to weaver-provided libraries, the interface of libraries provided in bundles is not enforced. However, we believe they still provide the same benefits.

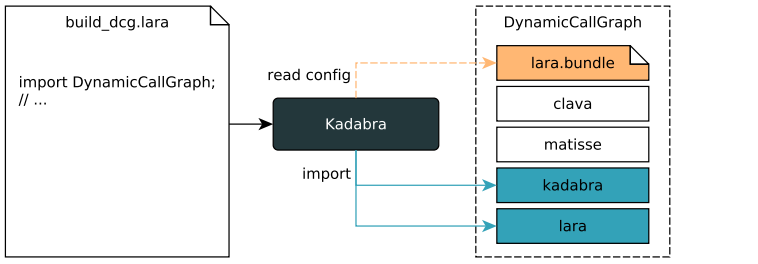

Figure 12: A LARA aspect importing a library provided in a bundle.

4.4. Common Language Specifications

We present one last set of techniques that improve aspect code reuse and support for multilanguage aspects. If besides the library APIs, weavers also share the part of the Language Specification used in an aspect (e.g., the call join point, in Figure 10), the aspect can be fully compatible between weavers, even in cases where the target language is not the same.

When designing a new weaver, one of the first steps is the definition of the Language Specification, which states the points in the code that may be selected, their attributes and the actions that can be applied over them. Internally, when developing our weavers, we have taken some care to make sure there is at least some overlap between the join points and attributes available for all weavers. In our case, it helps that all target languages are imperative and share similar constructs such as functions, function calls, loops, statements and expressions. This is the concept of *Common Join Point Model* presented in Section 3.

An important issue to keep in mind is that while we recommend trying to maintain a degree of compatibility between Language Specifications of different weavers, this is not enforced, nor do we think it should be. Besides sharing join points and attributes among weavers, which we have already seen in some examples, we can also consider two more techniques, generic weaver actions and join point aliases.

Below is a list with the subset of join points that are common to the weavers Clava, Kadabra and MATISSE, alongside a short description.

Application. The entire application. Useful to get information such as the type of the program (C/C++ in Clava) or to apply program-wide transformations.

File. A file or compilation unit. Used to add new functions or classes to a specific file. This is also the main modular unit in C and MATLAB programs.

Class. A class declaration. The main modular unit for JAVA programs. Used to change and extend or query a specific class.

Function. A function definition. Used to perform several operations at the function level, which include changing it, cloning it or introducing code at it's entry and exit points for instrumentation.

Loop. A loop, including its head and body. Used mostly for performance profiling and subsequent optimization.

If. An if statement. Used mainly to perform analyses, such as branch frequency and control flow.

Statement. A statement within a function. A base class for all other statements. Rarely used by itself except for some extensive analyses.

Call. A call to a function. Often used in instrumentation and in dynamically adaptable code.

Table 1: List of attributes that are shared in the common subset of join points of the weavers Clava, Kadabra and MATISSE.

Join Point	Attribute	Description
File	name	name of the file
	path	path to the file in the system
Class	name	name of the class
Function	name	name of the function
	type	return type of the function
Loop	controlVar	loop control variable
	isInnermost	whether the loop is innermost of a nest
	isOutermost	whether the loop is outermost of a nest
	nestedLevel	0 for outermost loops, +1 for each successive loop in the nest
Call	name	name of the function being called
Declaration	name	name of the declared variable
	type	type of the declared varaible
Variable	name	name of the variable being used
	reference	type of use of the variable: read or write
	type	type of the variable being used

Declaration. The declaration of a variable. This is often used to change the type or initial value of a variable.

Variable. A reference to a variable, which can be a read or a write. This is useful for instrumentation.

There are a three exceptions which are not reflected in the list. First, because C programs do not have classes, it is not possible to select them. However this does not result in an error in Clava, since it uses a shared model between C and C++. This means that no classes will be found and execution will resume normally. Second, it is not possible to select classes in MATISSE. Although MATLAB supports classes this feature was never a priority or a necessity for the work developed with this weaver, so it was never implemented. Finally, it is not possible to select declarations in MATLAB using MATISSE. Since MATLAB does not have a concrete concept of variable declaration, we decided to not implement such join point.

Table 1 shows, for each shared join point, which attributes are also shared among the weavers Clava, Kadabra and MATISSE. In this list there are three exceptions, the `type` attribute for the join points Function, Declaration and Variable are not part of the language specification provided by MATISSE. The reason is that MATISSE targets MATLAB, a language that is dynamically typed and does not provide type inference, resulting in very limited information about types that can be extracted at compile time.

```
aspectdef DeclareVariableInLoop
 select loop end
 apply
  exec DeclareVariable(Type.FLOAT,'X'+$loop.rank,3);
 end
 condition
  /*...*/
 end
end
```

Figure 13: An example of how a LARA action can be used for variable declaration.

4.4.1. Generic Weaver Actions

Since weavers have access to a complete IR of the original program, certain code transformations (e.g., loop transformations) may be easier to implement in the weaver itself than within LARA aspects. The language specification allows the definition of custom actions, which are implemented by the weaver developer. For instance, consider the case where we intend to declare a variable inside a given scope. Depending on the language, there can be several syntactic and semantic rules associated with this action. While this could be done with insertions of native code, a weaver action provides greater control and robustness (e.g., check whether there is a variable with the same name in the given scope, update the symbol table or warn the user if the declaration shadows another variable). Figure 13 shows an example of a LARA aspect that uses a weaver action (invoked using `exec`) to declare a variable inside the scope of all the loops in the given code.

If several weavers conform to the same standard for actions and provide the same semantics, we can have generic weaver actions, even if such actions are considered weaver specific, as mentioned in Section 2.1. For instance, if two weavers, one for JAVA and one for C, both implement the `DeclareVariable` action (seen in Figure 13) with the same interface and semantics, it is considered a generic action. These actions improve the development of language-independent aspects using LARA.

4.4.2. Join Point Aliases

In certain cases, there are points of interest in the code that are similar between languages, but that can have different names due to nature, history or conventions of the language (e.g., `function` in C vs `method` in JAVA). To increase compatibility between weavers, when specifying a language, the weaver developer can use *join point aliases*, which allows referring to the same join point using different names. For instance, in Kadabra, a JAVA weaver, `function` is an alias for `method`, which means that we can capture methods with any of the following `select` statements:

```
// these select statements are equivalent because function is an alias for method
select method end
// ... advice code here

select function end
```

```
// ... advice code here
```

With join point aliases, instead of forcing a single denomination to all languages, weavers can use their conventional denomination and still have compatibility with more generic aspects.

5. Evaluation

In this section we evaluate the approach using three weavers that target different languages: Kadabra[5] for Java, Clava[6] for C/C++ and MATISSE[7] for MATLAB. The aspect code used for this evaluation can be found in our online public repository[8]. In this evaluation we counted the number of logical lines of source code (i.e., SLoC) using LocMetrics[9], except for LARA aspects, where we used a custom tool that implements part of our heuristic for counting LARA lines of code[10].

5.1. Tooling Reuse

One of the objectives of this approach was to enable tooling reuse between weavers that target different languages, using LARA as their aspect language. We assume that when developing a new weaver, developers will most likely reuse existing grammars, parsers and ASTs for the target language, and this should not count towards the programming effort. For instance, both Clava and Kadabra use third-party compiler frameworks (Clang [17] and Spoon [13], respectively) to parse the code and obtain the AST. MATISSE reuses a custom parser and AST that was originally developed to translate MATLAB to C.

As already explained in Section 2, the LARA Framework is written in Java and contains a compiler, an interpreter and a tool to generate initial weaver implementations. The compiler parses LARA aspects and creates an intermediate representation in XML which can then be interpreted. The Weaver Generator is a tool that accepts a Language Specification and generates a skeleton weaver for that specification. The task of the weaver developer is to fill in the blanks and write the code that connects the points in the specification to the nodes in the AST of the source code. Using the Weaver Generator, and having a parser and AST for the target language, it is possible to have a working prototype in a few hours.

Figure 14 shows the logical source lines of code (SLoC), for the Kadabra, Clava and MATISSE weavers, divided into three components. *LARA* is the code size of the LARA framework, which is shared by all weavers. This represents the largest part of the total code and takes care of compiling and interpreting LARA aspects. The *Generated Code* slice represents LARA API code that is automatically produced by the Weaver Generator. This is the interface between a weaver implementation and the framework and consists

[5]https://specs.fe.up.pt/tools/kadabra
[6]https://specs.fe.up.pt/tools/clava
[7]https://specs.fe.up.pt/tools/matisse
[8]https://github.com/specs-feup/specs-lara/tree/master/2017 COMLAN
[9] http://www.locmetrics.com/
[10]https://web.fe.up.pt/ specs/projects/lara/doku.php?id=lara:docs:stats

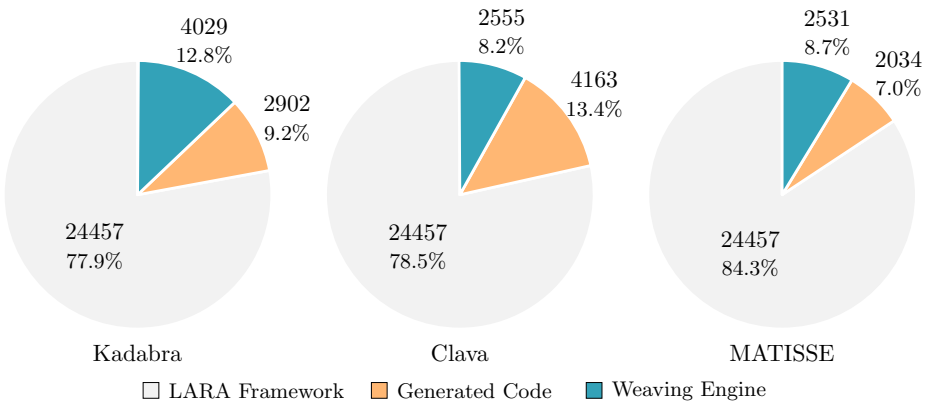

Figure 14: Lines of code for each weaver divided into components: LARA framework, auto-generated API and weaver engine.

mainly of abstract classes that the weaver developer needs to implement. Finally, *Weaving Engine* refers to code that is actually implemented by the weaver developer. This is a fairly small part of the total code and illustrates how much work one saves by using our framework when developing a weaver for a new target language. All weavers in Figure 14 have been in development for at least a year and support rich language specifications. Still, their size is a fraction of the LARA Framework, about an order of magnitude smaller.

The amount of code of the LARA Framework is indicative of the possible effort required to start an AOP approach from scratch. It is a medium-sized project and we estimate that, with its current features and field testing, represents well over an year of investment for a small team. Please note that the LARA framework is generic and does not have code specific to any target language. The LARA compiler, interpreter and generator used by these weavers are the same. Our approach allows a weaver developer to use an already existing target language compiler, significantly reducing the effort when developing a weaver from the start. For this reason, information about the compilers used is not included.

5.2. Impact of the Presented Techniques

Table 2 presents measurements of logical lines of source code (SLoC), disregarding blank lines and comments, taken from a set of LARA aspects for three concerns. Logging, timing code fragments and measuring energy consumption are three concerns we've used multiple times in previous research, with multiple target languages. We decided to develop libraries that encapsulate common operations performed to add the desired behavior. As these were general enough, we also decided to include them as part of standard LARA libraries and distribute them with our weavers. The left side of the table, under *User Aspects*, corresponds to the code that is written by a user of our weavers. We measured simple example aspects that apply these concerns on small native applications, which can somewhat account for the similarity between aspects. There are two versions of aspects, one version that uses no API and relies on the default features of LARA, which we consider our starting point, and one version that uses the API of the developed libraries. On the other hand, the right side of the table, under *Weaver Libraries*, corresponds to the code of the developed libraries.

Table 2: Logical lines of source code for both user-developed aspects and aspects distributed as a library by our weavers. Library code has shared parts (*Weaver-Agnostic*) and parts tied to each language (*Weaver-Specific*). User code is divided in aspects using the developed libraries (*With API*) and aspects weaving those concerns natively (*Without API*).

		User Aspects		Weaver Libraries	
Aspect	Language	With API	Without API (NSLoC %)	Weaver-Specific	Weaver-Agnostic
Logger	C	14	20 (50.0%)	110	126
	C++		17 (41.2%)		
	JAVA		9 (44.4%)	21	
	MATLAB		14 (50.0%)	33	
Timer	C	6	17 (35.3%)	114	28
	C++		15 (26.7%)		
	JAVA		10 (30.0%)	32	
	MATLAB		7 (28.6%)	23	
Energy	C	6	12 (25.0%)	32	39
	C++		12 (25.0%)		
	JAVA		15 (46.7%)	31	
	MATLAB		32 (71.9%)	28	

Again, it is divided in two, part of the code of the library is specific to the weaver, and the other part is shared between all implementations. The `Logger` library is used to incrementally build a message and then, at a specified join point, log it to a file or a console. The `Timer` library is used to time execution time around a join point (e.g. `call` or `loop`) and print the measured result. Finally, the `Energy` library works in a similar way but measures energy consumption (it relies on RAPL [18]).

Comparing user-developed aspects with and without using the provided libraries, we can see a generalized reduction in SLoC when using libraries. There is a single cases where the code using the API is larger and another where SLoC remains the same. For every other case, there is, on average, a reduction of 41.3% of SLoC. The largest reduction, when measuring energy consumption in MATLAB applications, represents a reduction of 81.3%. In this case, the MATLAB aspect without API has the code of a function needed to measure energy. All other versions were inserting calls to already existing native libraries to interface with RAPL. This explains why the aspect code for MATLAB is around twice as large as the code for other languages, which in turn explains the largest reduction in SLoC. There is a case where we increase SLoC by migrating to the API version, from 9 to 14. This happens with the JAVA version of `Logger`, and can be explained by the fact that the original aspect was inserting calls to an existing internal library used for

IO operations. In the context of logging, this code was responsible for writing text to a file in the system. The interface of our developed library ended up being more verbose than the original implementation of the concern for Kadabra/JAVA.

One important thing to note about these results, is that the code that uses the APIs is the exact same for every language. As an example, the aspect that instruments the code to measure execution time using the `Timer` library is the same for C, C++, JAVA and MATLAB. This is a consequence of the libraries providing functionality with a very high-level interface, where the user declares the intents, rather than specifying where and how to change behavior. The results presented in this table are mainly due to the technique *Generic Aspect Libraries*, presented in Section 4.

The weaver-specific portion of the weaver libraries (column *Weaver-Specific*) is responsible for weaving the same concerns as the code in the user aspects that does not use our API (column *Weaver-Specific*). However, the library code encapsulates the concerns over a well-defined interface and also performs additional checks. This has the effect of increasing the SLoC of this part of the code, which explains some of the large increases between the two.

In this table we can also see that a considerable part of the implementation of the libraries is shared between different weavers. This is possible because the Language Specifications of our weavers overlaps on the critical points needed to implement these concerns. Since our weavers all target imperative languages, the specifications share similar join points and attributes for common language constructs. This is the technique *Common Language Specifications*, as presented in Section 4. On average, 52.9% of the concrete implementations for each weaver is language agnostic and shared by all weavers. The rest of the implementation, the part which cannot be shared, differs mainly for two reasons. First, there are parts of the programs we wish to target, whose model is not common between weavers. This is normal and is not exposed to end users, as they only see the library interface. Second, different languages require different native code to be inserted to perform the same tasks.

In the column *Without API* we can see between parentheses the percentage of SLoC of the aspect that corresponds to native code insertions (*NSLoC %*). A significant part of the code of the aspects has native code since this is the main mechanism of providing additional behavior to the application. On average, 37.5% of all lines of code are written in the target language rather than in LARA. By moving to aspects that use the provided APIs (column *With API*) we no longer rely on direct insertion of native source code, at least on the user side. The implementation of the library inside each weaver makes use of insertions, but the user does not have to worry about these details. Instead, aspect developers can simply state their intentions using the high-level APIs for the presented concerns. Besides the obvious advantage of reducing the effort needed to write aspects, we also consider this leads to more robust programming, since the user inserts less native code (possibly unchecked by the weaver), which is more likely to contain errors.

The SLoC measurements in Table 3 correspond to example aspects that advise the code in order to instrument it for two purposes. One is for generating a call graph and the other for generating a report for

Table 3: Logical lines of source code for both user-developed aspects and user-developed libraries distributed as bundles. Library code has shared parts (*Weaver-Agnostic*) and parts tied to each language (*Weaver-Specific*). User code is divided in aspects using the developed libraries (*With Bundle*) and aspects weaving those concerns natively (*Without Bundle*).

		User Aspects		Bundle Libraries	
Aspect	Language	With Bundle	Without Bundle (NSLoC %)	Weaver-Specific	Weaver-Agnostic
DynamicCallGraph	C	9	27 (29.6%)	34	21
	C++				
	Java	9	142 (14.1%)	103	
	MATLAB	9	44 (25.0%)	45	
RangeValueMonitor	C	9	60 (36.7%)	93	20
	C++				
	Java	9	134 (15.7%)	121	
	MATLAB	9	71 (36.6%)	67	

the range of values taken by variables, both based on runtime information. Since these concerns were a bit more specific than the ones presented in Table 2, we decided not to distribute them with our weavers, but instead develop library bundles, as if they were user-developed libraries. This intends to measure the impact of the *Bundles* technique, presented in Section 3.

The conclusions for the results presented in this table are approximately the same as for the results of Table 2. For instance, using the API provided by the library bundle, SLoC is greatly reduced when compared to not using it. On average, we see a reduction of 83.7% in SLoC. These libraries still provide a high-level abstraction to deal with the described concerns. As with the previous examples, the original aspects (not using bundles) also contained native code insertions. On average, 24.6% of the SLoC of these aspects corresponds to native code insertion. The aspects using the libraries distributed as bundles no longer rely on the user writing native code in the aspect and specifying where it should be inserted. Despite the similarities, we still see some relevant differences between the results presented in each table.

There is also a considerable amount of shared aspect code in the implementations of the library. For each concern, on average 22.2% of the code of each weaver implementation is shared between all languages. The examples of Table 2 focused on more general tasks and the differences between weaver-specific implementations of the libraries were mostly due to native code insertions. On the other hand, the examples of Table 3 perform more specialized weaving with more complex logic. This code is not extracted to a library as easily, and we feel it justifies our initial decision of not including these concerns as standard aspect libraries distributed with all our weavers. This is also the reason why, in this case, the SLoC of user aspects not using API is closer to the SLoC of the weaver-specific implementation parts of the libraries. There is less to abstract in the original aspect code and less to share among implementations.

Although SLoC is the same per concern, we can see that SLoC for user aspects using bundles is discrim-

Table 4: Logical lines of source code for analysis aspects developed to target all languages support by our weavers.

Aspect	LoC
StaticCallGraph	10
DetectRecursion	16
StaticCodeReport	99

inated by language. With the exception of the aspects for C and C++, which are shared, the other aspects are different for each language. These changes result from small incompatibilities in the Language Specification of each weaver. The interface to these libraries expects to receive certain join points, the selection of which 1) cannot be abstracted and hidden inside the library, and 2) is slightly different among weavers.

Table 4 presents SLoC counts for another set of aspects, this time for static analysis of source code. They are considered analysis aspects, since they do not add additional behavior or change the structure of the original program, but rather collect static information about its source code. These are aspects we have developed over time and decided to make independent of the target language, since they were used in all weavers but with different implementations. `StaticCallgraph` builds a call graph based on the structure of the source code, which is outputted as a DOT file. `DetectRecursion` builds another graph representation of the program and checks if there are cycles, reporting their size if there are any. This aspect makes use of an external JavaScript graph library, which is loaded into the aspect. Finally, `StaticCodeReport` generates a report of the source code of the application with information about the structure of the code and the number of specific constructs (e.g., how many loops exist, discriminated by type). These aspects can be used by any of our weavers without changes. This is only possible since our weavers share, at least partially, join points, as well as their attributes and actions.

5.3. Enabling Higher-Level Aspects

In this section we presented several aspects that exemplify some of the opportunities this approach can enable. We do not think aspect developers should initially aim to write language-independent aspects, as they should not sacrifice expressiveness and legibility for this purpose. They should, however, follow a *coding – refactoring – library* cycle, as with any other programming language. In the case of aspects, this cycle should enable more generic aspects, written at a higher abstraction level that hides many complexities of the underlying target language and reduces the quantity of raw insertions of native code. We present tools and techniques that enable such a cycle, and that can be applied both at the aspect and target language level (e.g., aspect API and multilanguage aspect API, respectively).

In addition, if it is possible to standardize a set of common join points, actions and library APIs, we think this can enable further reuse, more specifically if performed at the weaver developer level. Keep in mind that we do not think it is necessary for all weavers to implement all common features; given the breath

of variety in programming languages, we consider partial (or even none) compatibility between weavers to be perfectly acceptable.

6. Related Work

The most well-known AOP approaches extend their target language with AOP concepts, e.g., AspectJ [2] extends JAVA and aims at providing better modularity for JAVA programs. AspectJ describes pointcuts lexically (e.g., `call(set*(..))`) and has a very mature tool support[11]. AspectJ join points are limited to object-oriented concepts, such as classes, method calls and fields, and several authors have proposed extensions to AspectJ. AspectC++ [3] is an AOP extension to the C++ programming language inspired by AspectJ, and uses similar concepts, adapted to C++. Both AspectJ and AspectC++ do not consider join points related to local variables, statements, loops, and conditional constructs. AspectMatlab [19] is another example of an AspectJ-inspired language, for MATLAB in this case. It adds some distinctive features related with MATLAB programs, such as the ability to capture multidimensional array accesses and loops.

6.1. Compiler Optimization Approaches

A number of approaches address concerns that are usually out of scope for traditional AOP (e.g., code transformations, compiler optimizations). CHiLL [20] is a declarative language focused on recipes for loop transformations. CHiLL recipes are scripts, written in separate files, which contain a sequence of transformations to be applied in the code during a compilation step. The PATUS framework [21] defines a DSL specifically geared toward stencil computations and allows programmers to define a compilation strategy for automated parallel code generation using both classic loop-level transformations (e.g., loop unrolling) and architecture-specific extensions (e.g., SSE). LARA takes a similar approach to source-to-source transformations with the use of actions, which are defined in the language specification and implemented by a weaver. One can select join points for optimization (e.g., loops), filter them based on their attributes and then apply transformation actions.

6.2. Term Rewriting Approaches

There are several term rewriting-inspired approaches for code analysis and transformation, such as Stratego/XT [22] and Rascal [23]. Term rewriting can also be used as the back-end component of AOP approaches [24, 25], as it provides a common framework for pattern matching and code transformation that can abstract from the target language. Such approaches require the complete grammar for each target language, which makes it possible to reuse the framework for different languages. Strategy reusability between languages may also be possible, as long as the grammars have common parts. LARA, on the other hand, promotes the usage of existing compiler frameworks (e.g. Spoon [13]) for parsing, analysis and transformations, and its join point model does not require a one-to-one correspondence to the provided intermediate

[11] Spring framework (https://spring.io) and Eclipse plugin (https://eclipse.org/aspectj)

representation. Another distinct feature of LARA, for the weaver developer side, is that weavers can be built in an incremental fashion, adding join points, attributes and actions as needed.

6.3. General Reuse In Aspect-Oriented Approaches

There are some approaches that focus on the issue of aspect reuse, but do not take into account multi-language support. For instance, ParaAJ [26] is an extension to AspectJ that uses the concept of parametric aspects to avoid bounding an aspect to a specific class, type or method. This allows greater reuse of the aspect code, and has been used to overcome some of the shortcomings of AspectJ related to its lack of reuse and parameterization, e.g., in the applicability of software patterns [27]. Another extension to AspectJ, Meta-AspectJ [28], offers a language that generates AspectJ code based on templates and automatic type inference for AspectJ constructs. Because this approach is also an extension to JAVA, it can use any of its features to parameterize the generated AspectJ code, e.g., reflection. CaesarJ [29] is a different AOP language that is based on the idea of developing aspects as reusable components. By relying on Object-Oriented Programming concepts, the approach treats aspects as classes and is able to control how they are applied. Wrappers are able to dynamically extend objects with new behavior in a way that promotes reuse while solving the problem of integrating both structural and behavioral changes.

LARA was designed with aspect reuse in mind and includes the concept of aspect definitions as modular units. These units, `aspectdef`s, can have parameters and return values, and work as procedures which can be called from within other aspect definitions. Hence, LARA has had support for this type of aspect reuse since its inception.

6.4. Multilanguage Approaches

LARA has been inspired by many AOP approaches, including AspectJ and AspectC++, but differs from these efforts in several ways. Unlike most approaches, LARA has been designed so that it is decoupled from a specific target language. In a similar way, Jackson and Clarke [14] envision a multilanguage approach, SourceWeave.NET, using an XML AOP language in the context of the .NET framework. This approach is tied to .NET and every new language one wants to support needs a parser that builds a CodeDOM graph, the representation expected by SourceWeave.NET. To alleviate this problem, one can use approaches such as Wu et al. [30], which explore component-based parsing. These can be used to promote reuse and composition at the parsing-level, by defining parsers for each component of the language (e.g., expression, statement, class). SourceWeave.NET, by having the same intermediate representation, allows the usage of reusable aspects, agnostic to the target language. However, the join points that one can select are already defined, coarse-grained and cannot be changed. These characteristics impose a limitation on what can be exposed from the target language and captured within aspects.

Weave.NET [31] provides a language-independent approach for .NET languages that weaves existing components written in any supported .NET language into the original application. These components and the application need not be in the same language. This approach shares many similarities with SourceWeave.NET

except for two main differences. First, Weave.NET works at the assembly level, over the Common Language Infrastructure of .NET. Second, Weave.NET's original and concern languages can be different and freely intermixed. Aspect specifications are also described as XML scripts that have a reference to an external implementation of the additional behavior to be included. The programming and join point model follow that defined by AspectJ.

UniAspect [16] targets shared application components in different languages and uses a common representation of the components to apply the aspects. It provides support for languages such as C, Java, C#, JavaScript and Python, and translates programs in each of these languages into its Unified Code Model, on which the weaving is performed. UniAspect keeps a similar syntax to AspectJ and introduces `@` annotations for identifying the target language. Inside a single aspect, one can write advice code for multiple languages by marking each code fragment with the corresponding `@` annotation.

Compose* [15] is a language-independent AOP framework focused on the Composition Filters model. It enables the specification, agnostic to the target language, of message filters and dispatch mechanisms which have both structural and behavioral effects. Because the language used to specify concerns was developed with concepts such as messages and entities in mind, the specification of a concern is completely independent of programming language and can be reused to target different languages. With this approach one can indicate where and under what conditions message passing is altered, and additional behavior is provided by the user separately in source files of the target language. This approach uses a common intermediate representation, mostly focused on the structure of the program, and makes no assumptions about front-ends, which means that any parser and compiler can be changed to generate the intermediate representation that Compose* expects. The filters resulting from the concern specification are translated to a language-independent control flow model which is then converted to specific target languages by language-specific weavers, which are also responsible for weaving the code in final application.

Another multilanguage approach is presented by Gray and Roychoudhury [24]. They make use of the DMS [32] framework and its Rule Specification Language (RSL) to define new join point languages (similar to AspectJ) and to target multiple languages. RSL is agnostic to the target language and is used to define rules to manipulate the internal representations that are generated by the parsers for specific languages. User-developed aspects are translated to parameterized RSL rules. For every new language that one intends to target, the weaver developer needs to develop tools such as lexer, parser and code generator. Furthermore, the weaver developer needs to define implementations of RSL for the join point language for the new target language. Compared to our LARA approach, one of the drawbacks is that, while the approach supports multiple languages, each aspect is bound to a specific target, since the user needs to write native code in the advice. In more recent work [25], the authors use metamodels for both the aspect language (front-end) and for RSL rules (back-end), coupled with model transformation descriptions (with ATL [33]) to transform from one model to the other. This allows a controlled flow from an input aspect to the automatically generated low-level RSL transformation rules. Still, both the aspect languages and the generated RSL transformations

are heavily tied to a specific target language, meaning that while there is a lot of tooling reuse in the framework, there is very little that can be reused in an aspect from one language to the other.

6.5. Extendability of Multilanguage AOP Approaches

In this subsection we compare the presented multilanguage approaches across two dimensions we consider important to the scope of this paper, extendability at the language specification level and extendability at the weaver level.

6.5.1. Language Specification Extensions

Compared to other approaches, LARA provides more flexibility in the join point model, which is based on composable select expressions (similar to functional queries [34]). Hence, LARA supports arbitrarily complex join point hierarchies, including different models of join points, e.g., MATISSE includes annotation-based join points based on comments. The way LARA supports attributes and actions is conceptually similar to the variables *thisJoinPoint* and *tjp* used in AspectJ and AspectC++, respectively, which contain meta-information related to the join point. However, attribute and action information in LARA are specified in the Language Description (attribute and action models, respectively) and can be extended by the weaver developer. This means that different languages can have substantially different join points and attributes while still 1) maintaining a common subset and 2) using the same aspect language, LARA.

In SourceWeave.NET [14], Weave.NET [31] and UniAspect [16], the points of execution one can target are closely tied to the underlying framework so extending this model is not an easy task and cannot be performed on a language basis. Similarly, the set of execution points one captures with Compose* [15] is tied to the Composition Filters model and cannot be extended.

The approaches of Gray and Roychoudhury [24] and Roychoudhury et al. [25] allow different aspect languages for different target languages, which means the user interface is a bit more expressive in relation to the target language, but there is no aspect reuse between languages.

Our approach stands on a middle ground where we allow sharing common language specifications without actually enforcing them. Therefore, for some concerns, one can actually develop a single aspect that is used to target multiple languages.

6.5.2. Weaver Extensions

With *weaver extensions* we mean adding a support for weaving a new language with the existing framework. This makes sense only for approaches that are not an extension of their target language, i.e., approaches such as AspectJ are bound to Java and it does not make sense to consider adding support for new languages.

SourceWeave.NET [14] uses a common intermediate representation for all languages it supports. In order to add support for a new language, a weaver developer needs a new parser that can convert the target language into the used representation, CodeDOM. At the back-end, support is also needed with code generators and/or compilers that can take a CodeDOM representation and generate assembly code. In

regard to adding support for new languages, Weave.NET [31] needs less work. Because it works at the CLI level, a compiler that transforms the target application into a .NET internal representation is sufficient. This seems simpler but may be more restrictive in the languages that can actually be added, as it may not be straightforward to compile any language to CLI code representations.

In order to extend UniAspect [16], one needs to implement the front- and back-ends for the new target language according to their Unified Code Model (UCM) and Unified Code Object representations. This means extending the underlying UNICOEN framework with support for 1) parsing the new language and building the UCM representation, and 2) taking a UCM representation and generating back the source code for the new target language.

Compose* [15] can also be extended with support for new languages by using any existing parser or compiler for the desired language and implementing a *type harvester*. This basically collects type and structural information from the parser and provides it in the format needed by the Core, the language-independent part of Compose*. At the back-end level, the weaver developer needs to implement a weaver, which will translate the language-independent representation into the target language and inject it into the application. If a new type of language is being considered, the developer may also need to augment the Core component with the concepts of entity and message for that language. As an example, entities are object instances in JAVA and files in C, while messages are method calls in JAVA and function calls in C.

As for the approaches of Gray and Roychoudhury [24] and Roychoudhury et al. [25], if the needed tooling doesn't exist in DMS, the weaver developer needs to add support for a new language domain. Following this, the developer may want to develop a new aspect language that is closely tied to the target or reuse an existing one if the join points needed are already covered. Additionally, the developer needs to define a new metamodel for the aspect language, if a new one is developed, in the approach of Roychoudhury et al. [25]. Finally, the developer needs to add support at the back-end. In the former approach [24], some ad-hoc conversions methods from the aspect language to RSL rules that target the new language are needed. The latter approach needs the development of ATL rules to convert models from the aspect language into models of RSL rules, which the RSL back-end will transform into RSL code.

In our approach, a new weaver is needed to add support for a new language. The weaver developer starts by writing a Language Specification, which defines the join points that are available and how they can be captured, their attributes, as well as the actions one can perform over them in order to advice the application. Alternatively, the developer can also take one existing specification and change it to suit the needs of the new language. Besides reducing the amount of work, this has the added benefit of join point, attribute and action sharing, which leads to reuse in the aspect code, greatly enhancing compatibility between languages. Then, the developer uses the Weaver Generator to automatically generate a skeleton implementation of the weaver based on the target language specification. This is the interface between the work of the weaver developer and the rest of the LARA framework. At this point, the developer needs a front-end that can parse the target language and build some kind of intermediate representation. Here, there is a lot of freedom, as the

developer can choose an existing tool, e.g., a source-to-source compiler, or build one from scratch. For the weaver to work, the developer needs to make the connections between the chosen intermediate representation and the LARA framework by implementing the skeleton methods.

6.6. Aspect Reuse in Multilanguage Approaches

Table 5 presents an overview of the multilanguage approaches regarding their implementations to achieve reusable aspects, for both pointcuts and advices, following the definitions of Section 3. For pointcuts, most mentioned approaches share, for their supported languages, a common join point model. For SourceWeave.NET [14], Weave.NET [31], UniAspect [16] and Compose* [15], this is a common execution model, since they are behavioral based. For our LARA approach, it is a shared syntactical and semantic model, since LARA follows a structural approach and it is possible to capture semantic join point attributes based on source code properties. This sharing of models is encouraged but not enforced, as it would limit individual implementations for specific languages. In the case of Gray and Roychoudhury [24] and Roychoudhury et al. [25], the aspect language is translated to language-specific RSL rules. Therefore it is tied to a specific language and there is no possibility of aspect reuse between languages.

As for advices, SourceWeave.NET [14], Weave.NET [31], Compose* [15] and LARA rely on an interface implementation while UniAspect [16] and LARA use multiversioning. As mentioned before, UniAspect uses annotation to mark pieces of code as advice code for specific target languages. SourceWeave.NET and Weave.NET both use the interface concept since, in the XML descriptors, the user specifies the name of the class and method that have the advice code, but the code itself is defined externally. Compose* works similarly, as in the concern specification the user writes the entity and message names, the code of which still needs to be provided separately from the concern. As for our LARA approach, we expose the interface of libraries for specific concerns and then let weaver developers implement them for concrete target languages and distribute the implementation with the weavers. For advices, the approaches of Gray and Roychoudhury [24] and Roychoudhury et al. [25] do not provide reuse of aspect code.

Our presented approach and UniAspect [16] follow a multilanguage implementation in which aspects are self-contained and can, therefore, be used directly to target multiple languages. On the other hand, with SourceWeave.NET [14], Weave.NET [31] and Compose* [15] one needs to write native code, separate from the aspect code and this will be weaved into the application according to specified strategy. With the works of Gray and Roychoudhury [24] and Roychoudhury et al. [25], the advice part of the aspect is written in the target language and encapsulated in the aspect itself. This means that, while we consider all these approaches to be multilanguage, with the first five, one may target multiple languages with a single aspect, while with the last two, one needs to change the aspect code to reflect the new target language.

Table 5: Classification of the multilanguage approaches according to the possible implementations for reusable pointcuts and advices, presented in Section 3.

Approach	Pointcut	Advice
SourceWeave.NET [14]	common (behavioral)	interface (descriptor)
Weave.NET [31]	common (behavioral)	interface (descriptor)
UniAspect [16]	common (behavioral)	multiversioning (annotation)
Compose* [15]	common (behavioral)	interface (messages)
Gray and Roychoudhury [24]	n/a	n/a
Roychoudhury et al. [25]	n/a	n/a
LARA	common (structural)	interface (library)
	common (semantic)	multiversioning (bundle)

7. Conclusions

This paper focused on the use of LARA in the context of aspect composition and multilanguage targeting. We presented the LARA framework, composed by the LARA language, its compiler and interpreter and a weaver generator. We showed how the initial design of LARA and its most recent developments contribute to highly modular and composable aspect code, which we believe is the base for multilanguage support. We enumerated several possible options for aspect composition and reuse at both the pointcut and advice level, and then presented our techniques to achieve these goals. The techniques include the development of weaver- and user-side libraries with clearly defined interfaces and concrete implementations for each target language, and also sharing sets of common join points and attributes across different languages. We discussed the impact of the proposed techniques, and showed that our approach is able to enhance multilanguage support and, at the same time, to improve aspect code reuse, while also enabling more concise and less error-prone aspects. Furthermore, we also showed that, by using the components of the LARA framework, we can significantly reduce the effort needed to develop new weavers and provide support new target languages.

As future work we plan to continue the development of aspect libraries, extending and improving the implementations of existing ones, as well as adding new concerns as needed. We will also focus on improving the existing bundle system to allow greater flexibility for the user when implementing libraries that share code. Finally, we intend to further explore the opportunities provided by a single AOP approach in projects that use several target languages and customized tool flows. We intend to extend the LARA framework to support dynamic actions, as a way to instruct the weaver that the LARA code in those actions should be executed during application runtime, instead of relying only on insertion of native code.

Acknowledgments

This work was partially funded by the ANTAREX project through the EU H2020 FET-HPC program under grant no. 671623. João Bispo acknowledges the support provided by Fundação para a Ciência e a Tecnologia, Portugal, under Post-Doctoral grant SFRH/BPD/118211/2016.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: European conference on object-oriented programming, vol. 1241, Springer, 220–242, 1997.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, in: European Conference on Object-Oriented Programming, Springer, 327–354, 2001.

[3] O. Spinczyk, A. Gal, W. Schröder-Preikschat, AspectC++: An Aspect-oriented Extension to the C++ Programming Language, in: Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications, CRPIT '02, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 53–60, 2002.

[4] J. Fabry, T. Dinkelaker, J. Noyé, E. Tanter, A Taxonomy of Domain-Specific Aspect Languages, ACM Comput. Surv. 47 (3) (2015) 40:1–40:44, ISSN 0360-0300.

[5] J. M. P. Cardoso, T. Carvalho, J. G. F. Coutinho, W. Luk, R. Nobre, P. Diniz, Z. Petrov, LARA: an aspect-oriented programming language for embedded systems, in: Proceedings of the 11th annual international conference on Aspect-oriented Software Development, ACM, 179–190, 2012.

[6] J. M. P. Cardoso, J. G. F. Coutinho, T. Carvalho, P. C. Diniz, Z. Petrov, W. Luk, F. Gonçalves, Performance-driven Instrumentation and Mapping Strategies Using the LARA Aspect-oriented Programming Approach, Softw. Pract. Exper. 46 (2) (2016) 251–287, ISSN 0038-0644.

[7] P. Pinto, T. Carvalho, J. Bispo, J. M. Cardoso, Lara as a Language-Independent Aspect-Oriented Programming Approach, in: Proceedings of the Symposium on Applied Computing, ACM, 1623–1630, 2017.

[8] D. August, K. Pingali, D. Chiou, R. Sendag, J. Y. Joshua, et al., Programming multicores: Do applications programmers need to write explicitly parallel programs?, IEEE Micro (3) (2010) 19–33.

[9] E. Hilsdale, J. Hugunin, Advice weaving in AspectJ, in: Proceedings of the 3rd international conference on Aspect-oriented software development, ACM, 26–35, 2004.

[10] I. Nagy, On the Design of Aspect-Oriented Composition Models for Software Evolution, Ph.D. thesis, Faculty of Electrical Engineering, Mathematics & Computer Science, University of Twente, Netherlands, 2006.

[11] J. M. Cardoso, T. Carvalho, J. G. Coutinho, R. Nobre, R. Nane, P. C. Diniz, Z. Petrov, W. Luk, K. Bertels, Controlling a complete hardware synthesis toolchain with LARA aspects, Microprocessors and Microsystems 37 (2013) 1073 – 1089, special Issue on European Projects in Embedded System Design: EPESD2012.

[12] J. M. P. Cardoso, J. G. de F. Coutinho, R. Nane, V.-M. Sima, B. Olivier, T. Carvalho, R. Nobre, P. C. Diniz, Z. Petrov, K. Bertels, F. Gonçalves, H. van Someren, M. Hübner, G. Constantinides, W. Luk, J. Becker, K. Krátký, S. Bhattacharya, J. C. Alves, J. C. Ferreira, The REFLECT Design-Flow, Springer New York, New York, NY, 13–34, 2013.

[13] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, L. Seinturier, Spoon: A library for implementing analyses and transformations of java source code, Software: Practice and Experience 46 (9) (2016) 1155–1179.

[14] A. Jackson, S. Clarke, Sourceweave. net: Cross-language aspect-oriented programming, in: International Conference on Generative Programming and Component Engineering, Springer, 115–135, 2004.

[15] A. de Roo, M. Hendriks, W. Havinga, P. Durr, L. Bergmans, Compose*: a Language- and Platform-Independent Aspect Compiler for Composition Filters, 2, 2008.

[16] A. Ohashi, K. Sakamoto, T. Kamiya, R. Humaira, S. Arai, H. Washizaki, Y. Fukazawa, UniAspect: a language-independent aspect-oriented programming framework, in: Proceedings of the 2012 workshop on Modularity in Systems Software, ACM, 39–44, 2012.

[17] clang: a C language family frontend for LLVM, `https://clang.llvm.org/`, accessed: 2017-08-01, 2017.

[18] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, C. Le, RAPL: Memory power estimation and capping, in: 2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED), 189–194, 2010.

[19] T. Aslam, J. Doherty, A. Dubrau, L. Hendren, AspectMatlab: An Aspect-oriented Scientific Programming Language, in: Proceedings of the 9th International Conference on Aspect-Oriented Software Development, ACM, New York, NY, USA, 181–192, 2010.

[20] G. Rudy, M. M. Khan, M. Hall, C. Chen, J. Chame, A programming language interface to describe transformations and code generation, in: International Workshop on Languages and Compilers for Parallel Computing, Springer, 136–150, 2010.

[21] M. Christen, O. Schenk, H. Burkhart, Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), IEEE, 676–687, 2011.

[22] M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. A language and toolset for program transformation, Science of Computer Programming 72 (1-2) (2008) 52 – 70.

[23] P. Klint, T. Van Der Storm, J. Vinju, Rascal: A domain specific language for source code analysis and manipulation, in: Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on, IEEE, 168–177, 2009.

[24] J. Gray, S. Roychoudhury, A Technique for Constructing Aspect Weavers Using a Program Transformation Engine, in: Proceedings of the 3rd International Conference on Aspect-oriented Software Development, AOSD '04, ACM, New York, NY, USA, 36–45, 2004.

[25] S. Roychoudhury, J. Gray, F. Jouault, A Model-Driven Framework for Aspect Weaver Construction, in: S. Katz, M. Mezini, C. Schwanninger, W. Joosen (Eds.), Transactions on Aspect-Oriented Software Development VIII, Springer Berlin Heidelberg, Berlin, Heidelberg, 1–45, 2011.

[26] K. Aljasser, P. Schachte, ParaAJ: Toward Reusable and Maintainable Aspect Oriented Programs, in: Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91, ACSC '09, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 65–74, 2009.

[27] K. Aljasser, Implementing design patterns as parametric aspects using ParaAJ: The case of the singleton, observer, and decorator design patterns, Computer Languages, Systems & Structures 45 (2016) 1–15.

[28] D. Zook, S. S. Huang, Y. Smaragdakis, Generating AspectJ programs with meta-AspectJ, in: Generative Programming and Component Engineering, vol. 4, Springer, 1–18, 2004.

[29] I. Aracic, V. Gasiunas, M. Mezini, K. Ostermann, Transactions on Aspect-Oriented Software Development, chap. An Overview of Caesarj, Springer-Verlag, Berlin, Heidelberg, 135–173, 2006.

[30] X. Wu, B. R. Bryant, J. Gray, M. Mernik, Component-based LR parsing, Computer Languages, Systems & Structures 36 (1) (2010) 16–33.

[31] D. Lafferty, V. Cahill, Language-independent aspect-oriented programming, in: ACM SIGPLAN Notices, ACM, 1–12, 2003.

[32] I. D. Baxter, C. Pidgeon, M. Mehlich, DMS: program transformations for practical scalable software evolution, in: 26th International Conference on Software Engineering (ICSE), IEEE, 625–634, 2004.

[33] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A model transformation tool, Science of computer programming 72 (1) (2008) 31–39.

[34] M. Eichberg, M. Mezini, K. Ostermann, Pointcuts as functional queries, in: Asian Symposium on Programming Languages and Systems, Springer, 366–381, 2004.